

---

# Programación en C

## (Segunda Parte)

DATSI, FI, UPM

José M. Peña

`jmpena@fi.upm.es`

# Índice

---

- Estructura de un programa C.
- Variables básicas.
- Operaciones aritméticas.
- Sentencias de control.
- Arrays y Strings.
- Funciones.
- Estructuras de datos.
- Entrada/Salida básica.
- Ejemplos I.
- Modificadores de ámbito de las variables.
- Punteros y memoria dinámica.
- Operadores de bit.
- Preprocesador C y compilación.
- Librerías estándar.
- Ejemplos II.

---

# Programación en C

## Modificadores de Ámbito

# Modificadores de Variables

---

- La declaración de variables acepta los siguientes modificadores:
  - **static** (Local): El valor de la variable se conserve entre llamadas. Comportamiento similar a una variable global.
  - **register** : La variable es almacenada siempre (si es posible) en un registro de la CPU (no en memoria).
  - **volatile** : Un proceso exterior puede modificar la variable.
  - **const** : La variable no puede ser modificada.

# Modificadores de Variables

---

```
int una_funcion(int a, int b)
{
    static    char last;
    register  int  i;
    const     int  max=12;
    volatile long acc;

    . . .
}
```

# Modificadores de Variables (**static**)

---

```
void cuenta( )  
{  
    static int cnt=0;  
    printf( "%d\n" , cnt++ )  
}
```

```
int main( )  
{  
    cuenta( ) ; cuenta( ) ; cuenta( ) ; cuenta( ) ;  
    return 0 ;  
}
```

Salida:

0  
1  
2  
3

# Modificadores de Variables (const)

---

```
const int max=10;
int letra(const char* text, char l)
{
    int i,acc=0;
    for(i=0;i<max && text[i];i++)
        if(text[i]==l)
            acc++;
    return acc;
}
```

# Modificadores de Funciones

---

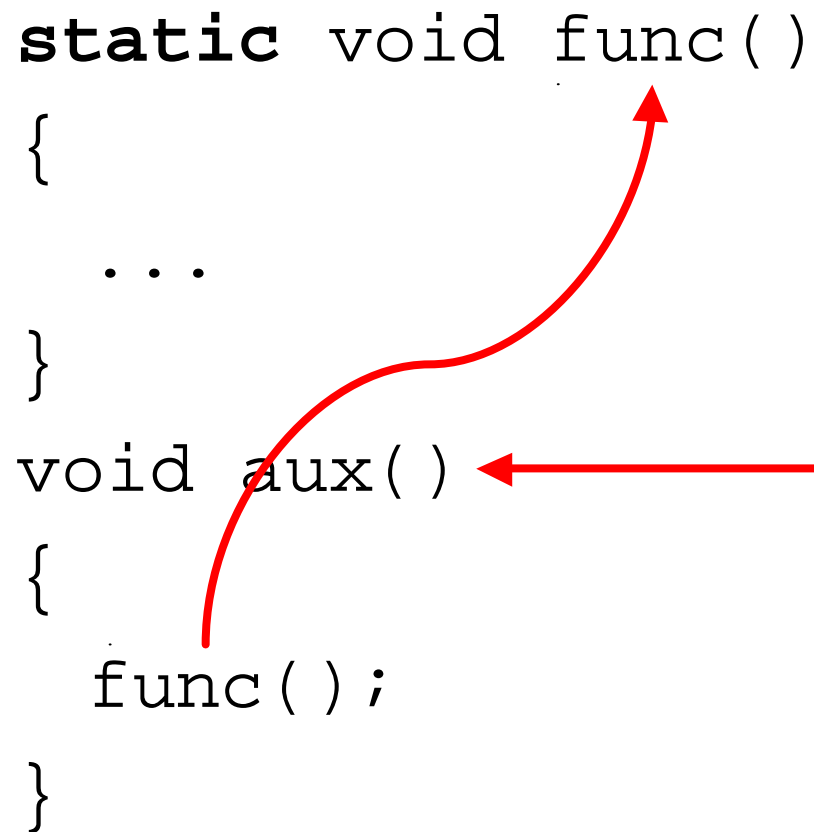
- Las funciones también pueden ser declaradas con ciertos modificadores:
  - **static** : Restricción de enlace. Sólo se puede usar dentro del mismo fichero (también variables globales).
  - **extern** : La función o variable se encuentra declarada pero no definida. Usada con variables globales.
  - **inline** : La función es expandida íntegramente al ser invocada. No hay un salto a la función. Incrementa la eficiencia y aumenta el tamaño del código.



# Modificadores de Funciones

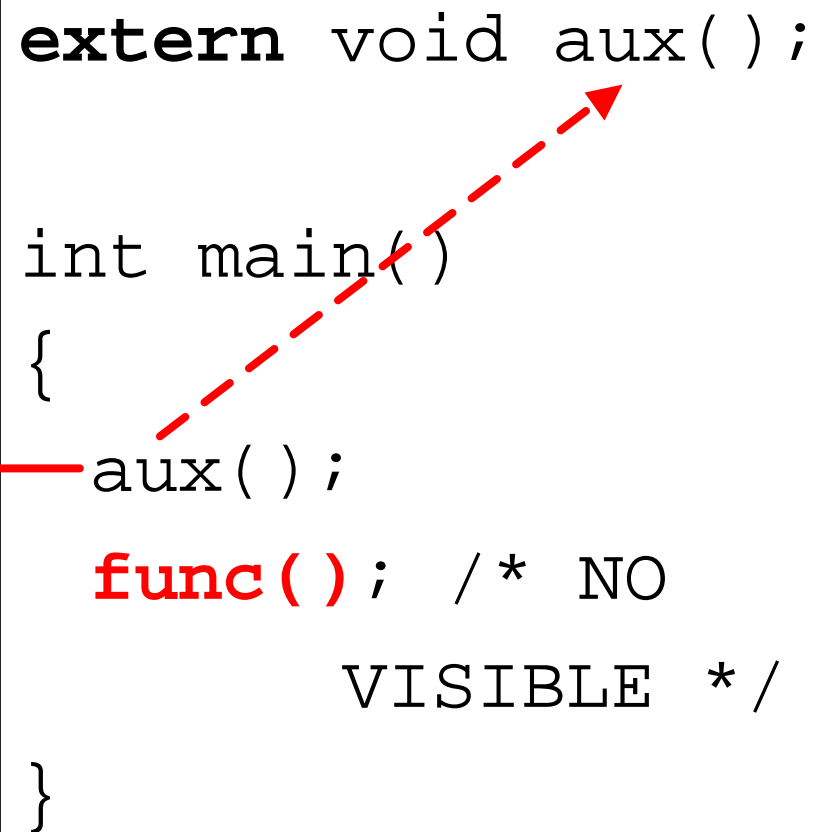
## Fichero1.c:

```
static void func()  
{  
    ...  
}  
void aux()  
{  
    func();  
}
```

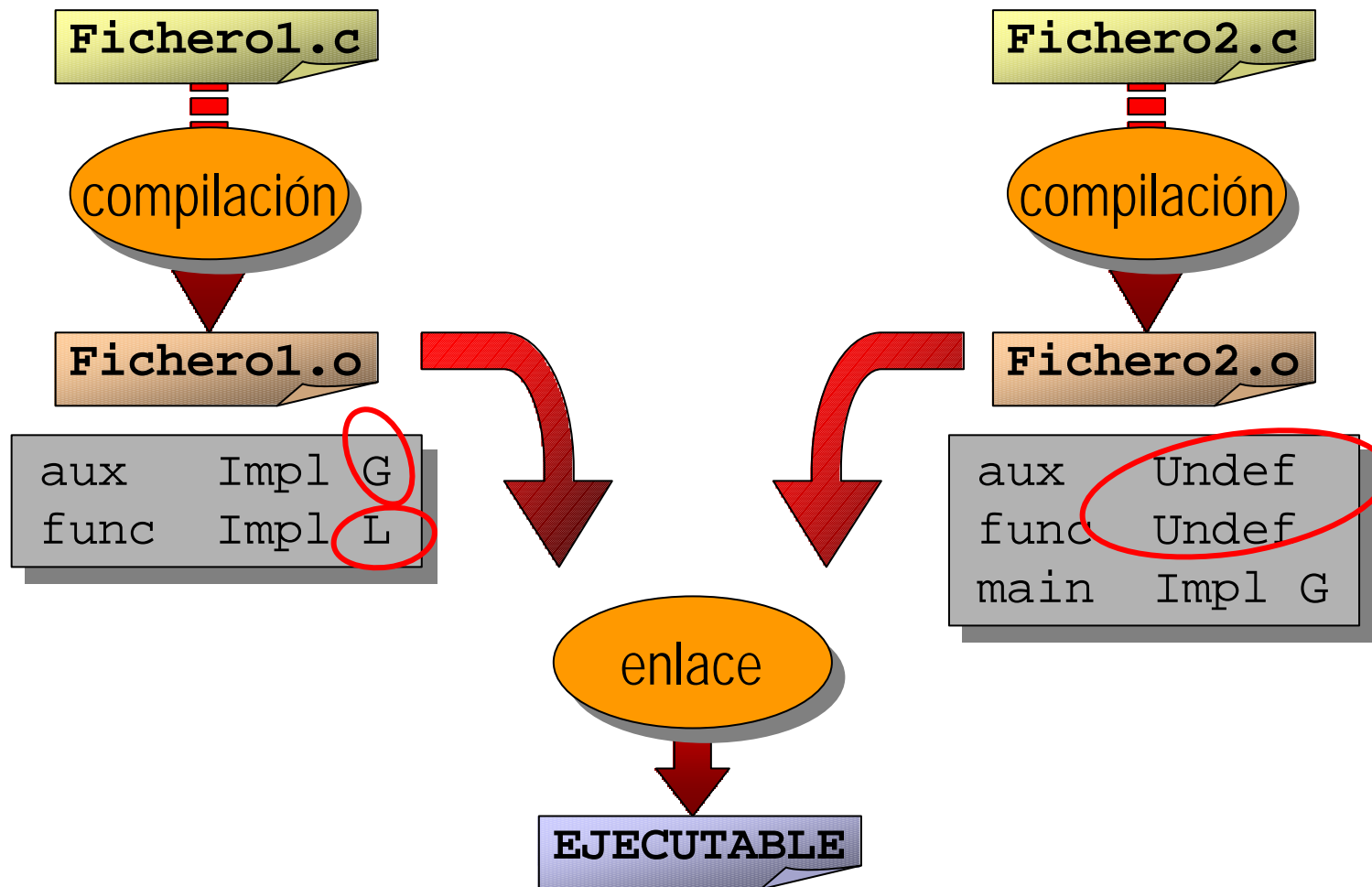


## Fichero2.c:

```
extern void aux();  
  
int main()  
{  
    aux();  
    func(); /* NO  
                VISIBLE */  
}
```



# Modificadores de Funciones



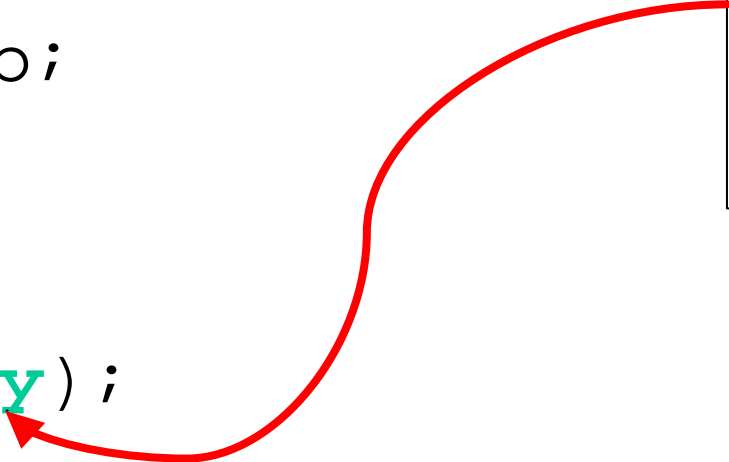
# Modificadores de Funciones

---

```
inline int max(int a, int b)
{
    if(a>b)
        return a;
    else
        return b;
}
```

....

```
x=max(x+1, y);
```



```
{
    if(x+1>y)
        x=x+1;
    else
        x=y;
}
```

---

# Programación en C

## Punteros y Memoria Dinámica

# Aritmética de Punteros

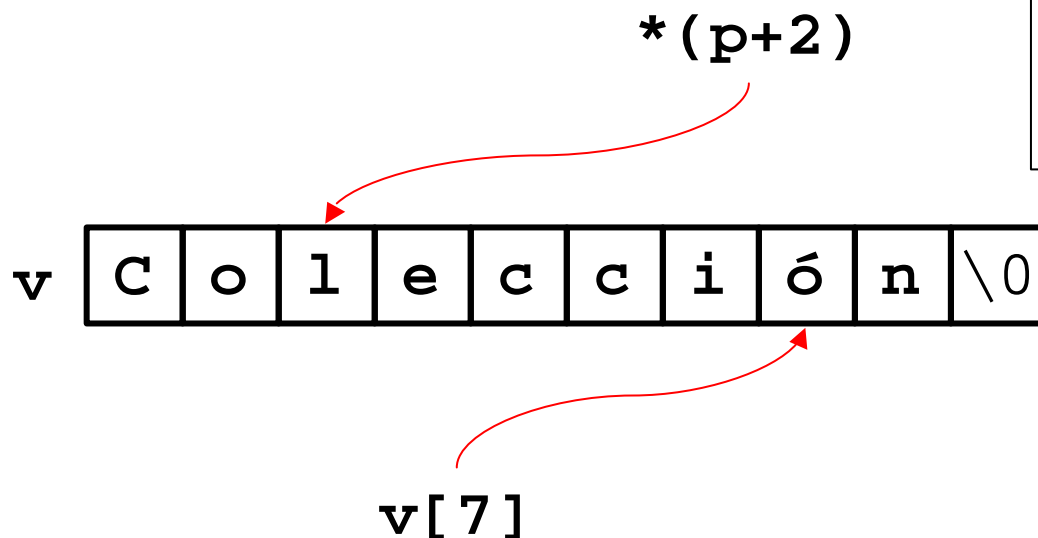
---

- Las variables de tipo puntero soportan ciertas operaciones aritméticas.

```
char v[]="Colección";
```

```
char *p=v;
```

```
for(p=v;*p;p++)  
{  
    printf("%c",*p)  
}
```



# Aritmética de Punteros

---

Las operaciones soportadas sobre punteros son:

- Suma y resta de valores enteros (+, -, ++ y --).
- Comparación y relación (<, >, <=, >=, == y !=).
- Valor booleano (comparación con **NULL**).

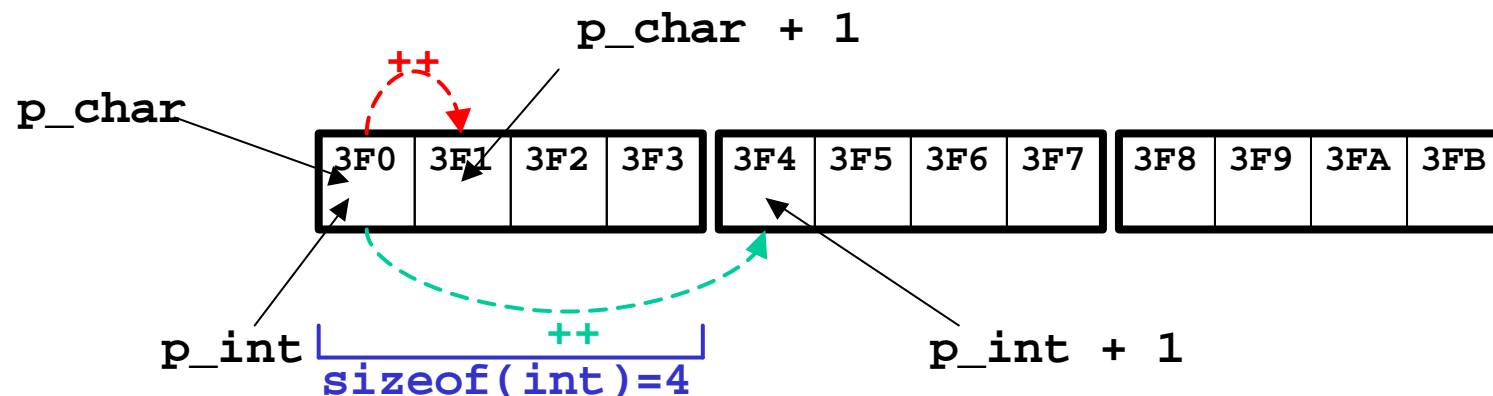
```
void copiar(char* dest, const char* orig)
{
    if(orig && dest)
        while(*orig)
            *dest++ = *orig++ ;
}
```

# Aritmética de Punteros

---

Las operaciones de suma o resta sobre punteros modifican el valor del dependiendo del tipo del puntero:

```
int*    p_int; char* p_char; p_int=p_char;  
p_int++; /* Suma sizeof(int) */  
p_char++; /* Suma sizeof(char) */
```



# Punteros a Funciones

---

Mecanismo para pasar funciones como argumento:

```
char (*f)(int,int);
```

**f** es un puntero a una función que devuelve un **char** y recibe dos enteros como argumento.

A un puntero a función se le puede asignar como valor cualquier identificador de función que tenga los mismos argumentos y resultado.



# Punteros a Funciones

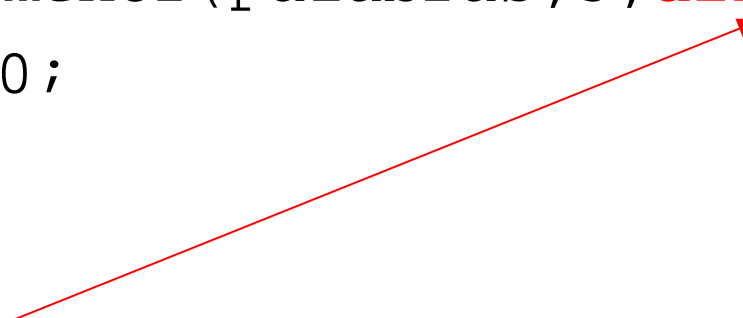
---

```
char* menor (char** text,
             int    tam,
             int     (*compara) (char*, char*))
{
    int    i;
    char*  min=text[0];
    for(i=1;i<tam;i++)
        if( *compara(minor,text[i]) )
            min=text[i];
    return min;
}
```

# Punteros a Funciones

---

```
int main()  
{  
    char *palabras[] = { "hola", "casa", "perro",  
                        "coche", "rio" };  
    printf( "Menor:%s",  
           menor(palabras, 5, alfabetico) );  
    return 0;  
}  
  
int alfabetico(char* a, char* b)  
{ .... }
```



# Memoria Dinámica

---

Además de la reserva de espacio estática (cuando se declara una variable), es posible reservar memoria de forma dinámica.

Funciones de gestión de memoria dinámica:

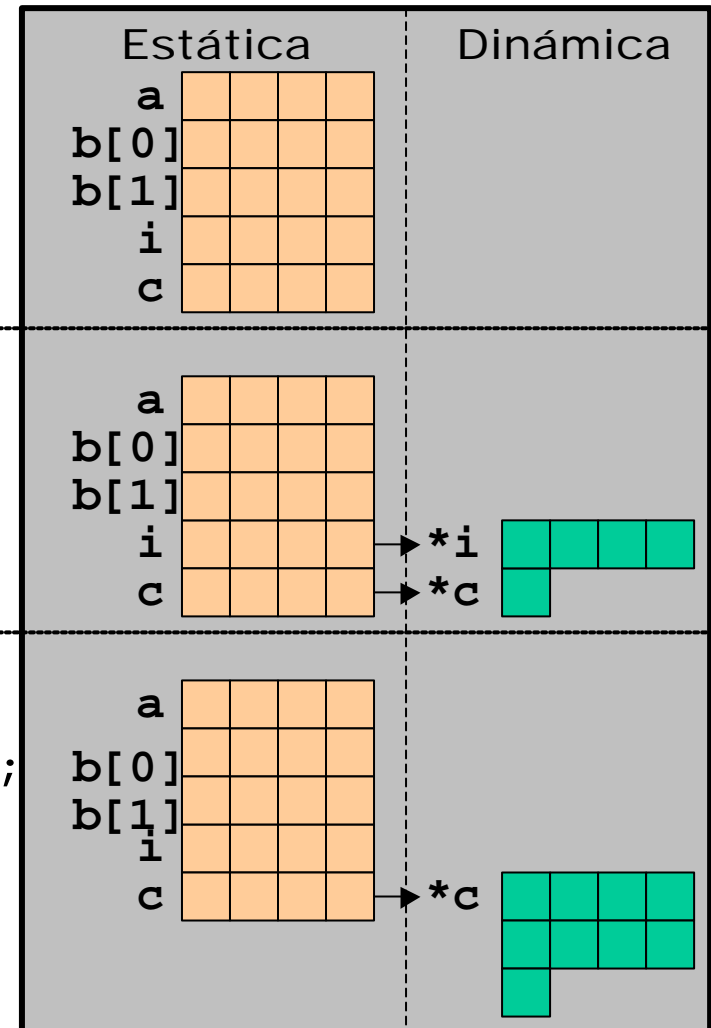
- **`void* malloc(size_t)`**: Reserva memoria dinámica.
- **`free(void*)`**: Libera memoria dinámica.
- **`void* realloc(void*, size_t)`**: Ajusta el espacio de memoria dinámica.

# Memoria Dinámica

```
int    a,b[2];  
int*   i;  
char*  c;
```

```
i=(int*)malloc(sizeof(int));  
c=(char*)malloc(sizeof(char));
```

```
free(i);  
c=(char*)realloc(c,sizeof(char)*9);
```



---

# Programación en C

## Operadores de Bit

# Operadores de Bit

---

Además de los operadores aritméticos y booleanos existen operadores numéricos a nivel de bit:

- AND: &
- OR: |
- XOR: ^
- NOT: ~
- Desplazamientos: << y >>

# Operadores de Bit

---

char a=48;      00110000 a

char b=19;      00010011 b

char x,y,z,w,t,s;

x=a & b;      00010000 x = 16

y=a | b;      00110011 y = 51

z=a ^ b;      00100011 z = 35

w=~a;      11001111 w = 207

t=a>>2;      00001100 t = 12

s=b<<3;      10011000 s = 152

# Uso de los Operadores de Bit

---

```
const char LECTURA  =1;
const char ESCRITURA=2;
const char EJECUCION=3;

char permisos=LECTURA | ESCRITURA;
permisos|=EJECUCION;
permisos&=~ESCRITURA;

if(permisos & EJECUCION)
    printf("Es ejecutable");
```

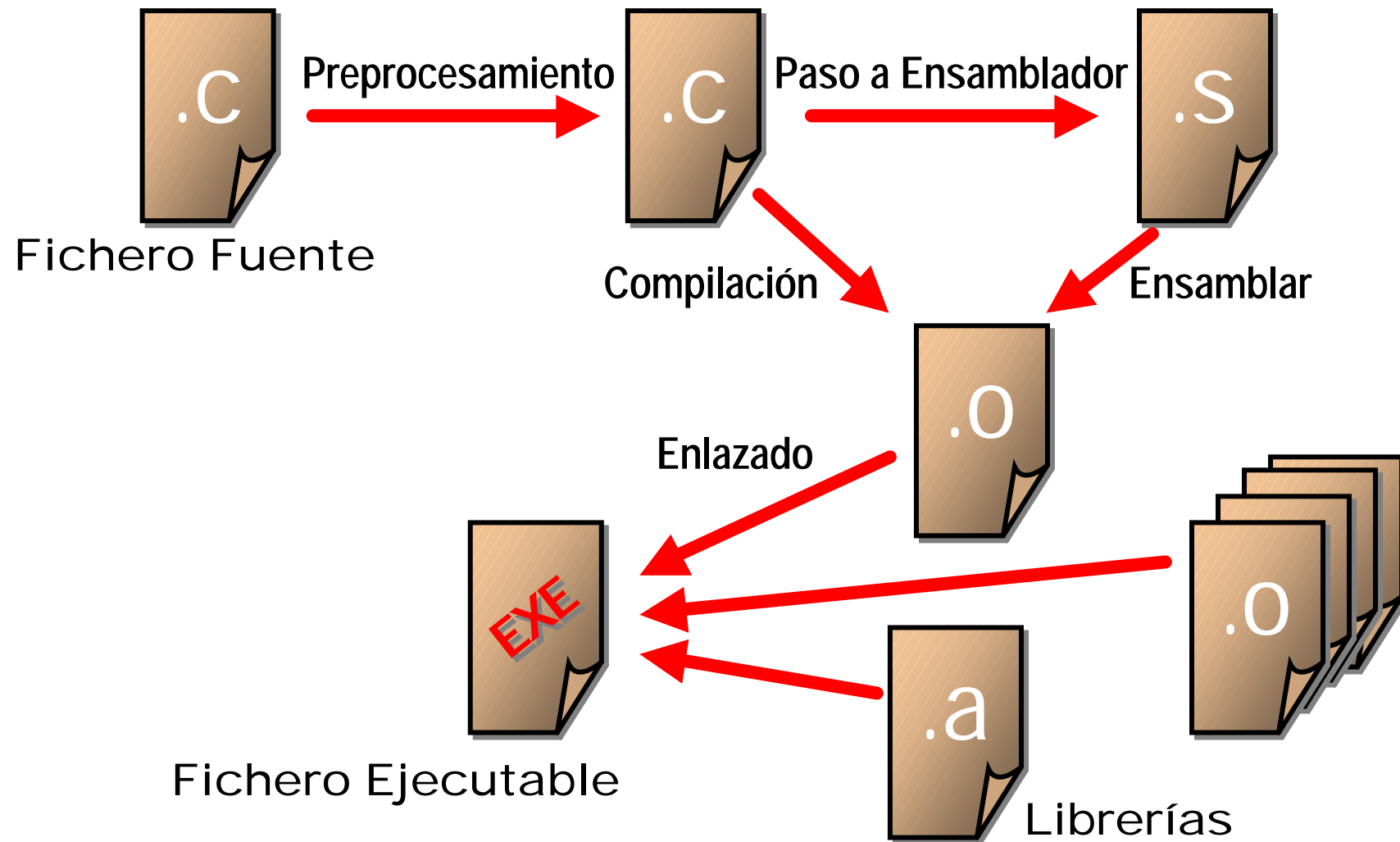


---

# Programación en C

Preprocesador y Compilación

# Fase de Compilación



# Directrices del Preprocesador

---

Son expandidas en la fase de preprocesado:

- **#define** : Define una nueva constante o macro del preprocesador.
- **#include** : Incluye el contenido de otro fichero.
- **#ifdef #ifndef** : Preprocesamiento condicionado.
- **#endif** : Fin de bloque condicional.
- **#error** : Muestra un mensaje de error

# Constantes y Macros

---

Permite asociar valores constantes a ciertos identificadores expandidos en fase de preprocesamiento:

```
#define variable valor
```

Define funciones que son expandidas en fase de preprocesamiento:

```
#define macro(args,...) función
```

# Constantes y Macros

---

```
#define PI          3.14
#define NUM_ELEM    5
#define AREA(rad)   PI*rad*rad
#define MAX(a,b)     (a>b ? a : b)
int main()
{
    int    i;
    float  vec[NUM_ELEM];
    for(i=0;i<NUM_ELEM;i++)
        vec[i]=MAX( (float)i*5.2, AREA(i) );
}
```

# Constantes y Macros

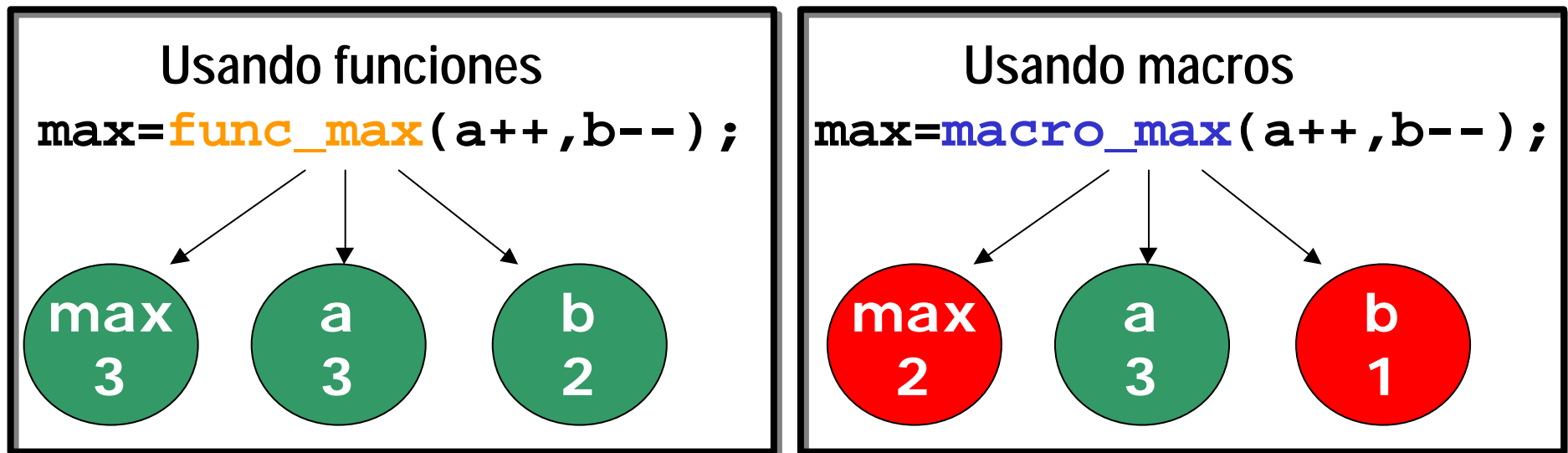
---

Tras la fase de prerprocesamiento

```
int main()  
{  
    int    i;  
    float  vec[5];  
    for(i=0;i<5;i++)  
        vec[i]=((float)i*5.2>3.14*i*i ?  
                (float)i*5.2 :  
                3.14*i*i);  
}
```

# Macros vs Funciones

```
int func_max(int a, int b)
{ return (a>b ? a : b); }
#define macro_max(a,b) (a>b ? a : b)
int a=2,b=3,max;
```

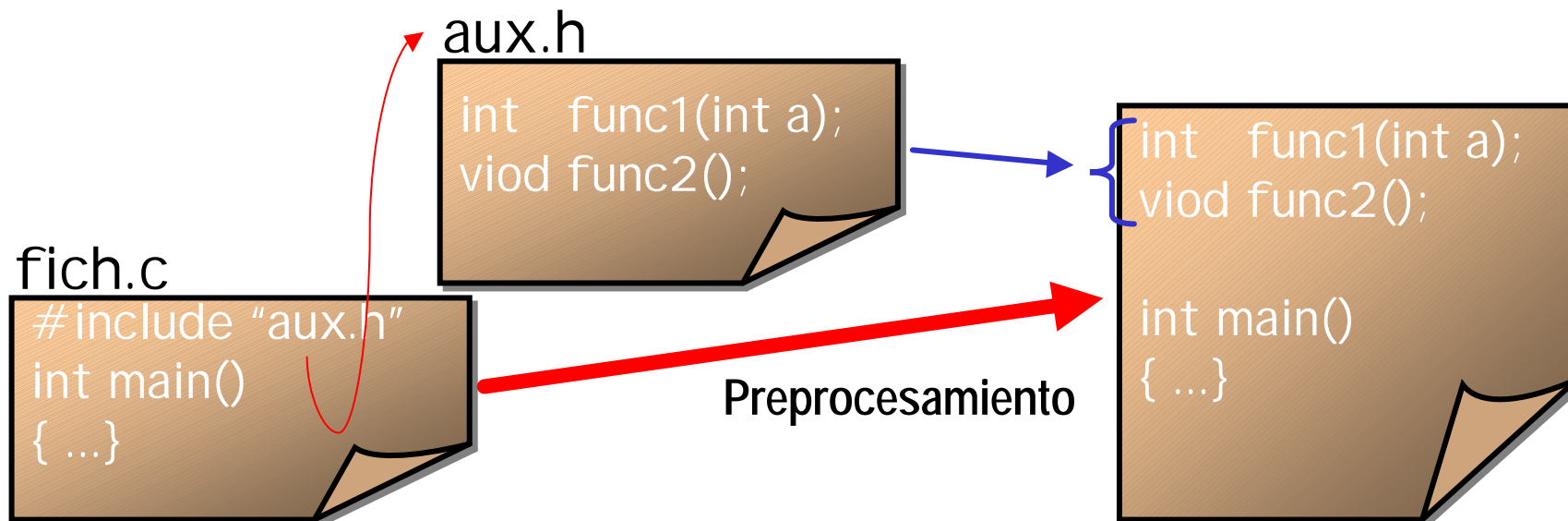


# Inclusión de Ficheros

Los prototipos de las funciones usadas por varios ficheros fuente se suelen definir en fichero de cabecera.

**#include <stdio.h>** Cabeceras del sistema.

**#include "mis\_func.h"** Ficheros de cabecera locales.





# Inclusión de Ficheros

---

La inclusión de ficheros esta sujeta a las siguientes recomendaciones:

- Por lo general los ficheros de cabecera tienen como **extensión .h**
- En los ficheros de cabecera **no se incluyen implementación** de funciones
- Las variables en un fichero de cabecera son **declaradas extern** y se encuentran declaradas en algún otro fichero **.c**

# Sentencias Condicionales

---

Para incluir código cuya compilación es dependiente de ciertas opciones, se usan los bloques:

```
#ifdef variable
```

```
<bloque de sentencias>
```

```
...
```

```
#endif
```

```
#ifndef variable
```

```
<bloque de sentencias>
```

```
...
```

```
#endif
```

# Ejemplo: Depuración

---

```
#define DEBUG
int main()
{
    int i,acc;
    for(i=0;i<10;i++)
        acc=i*i-1;
#ifdef DEBUG
    printf("Fin bucle acumulador: %d",acc);
#endif
    return 0;
}
```

# Ejemplo: Fichero de Cabecera

---

aux.h

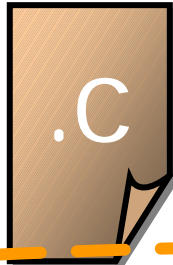
```
#ifndef _AUX_H_
#define _AUX_H_
<definiciones>
#endif
```

Evita la redefinición de  
funciones y variables

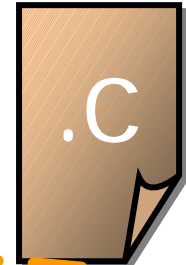
```
#include "aux.h"
#include "aux.h"
int main( )
{
...
}
```

# Enlace de Ficheros

```
extern int v;  
int main() {.h().}  
extern void k();  
static h() {.k().}
```



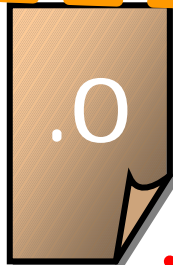
```
int k() {.1().}  
int l() {...}  
int v;
```



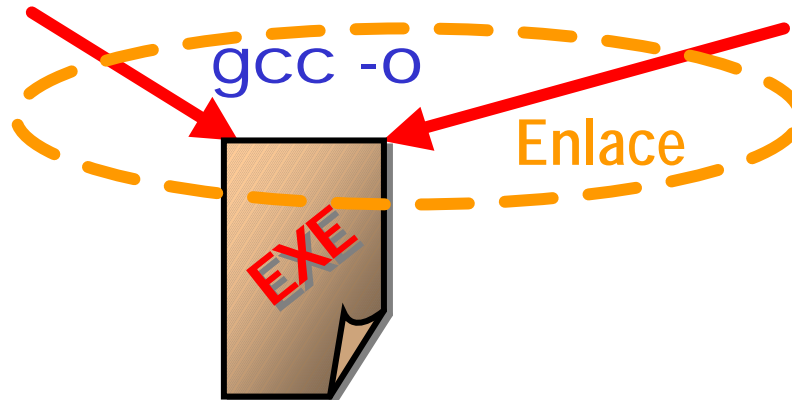
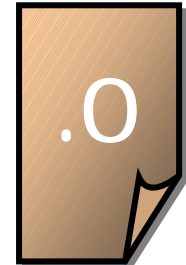
**Compilación**

`gcc -c` (indicated by red arrows pointing to the .O files)

3A00: f  
3A08: main  
3B12: h  
v <undef>  
k <undef>



1600: k  
17FF: l  
1812: v



---

# Programación en C

Librerías Estándar

# Manejo de Cadenas

---

- `char* strcat(char*,char*)`: Concatena cadenas.
- `char* strchr(char*,char)`: Busca un carácter.
- `int strcmp(char*,char*)`: Comparación de cadenas.
- `char* strcpy(char*,char*)`: Copia cadenas.
- `char* strdup(char*)`: Duplica una cadena.
- `int strlen(char*)`: Longitud de una cadena.
- `char* strncpy(int,char*,char*)`: Copia cadenas.
- `char* strncat(int,char*,char*)`: Concatena.
- ...

# Manejo de Buffers

---

- `void* memcpy(void*, void*, int)`: Copia memoria.
- `void* memmove(void*, void*, int)`: Copia memoria.
- `int memcmp(void*, void*, int)`: Compara memoria.
- `void* memset(void*, int, int)`: Rellena memoria.
- `void bzero(void*, int)`: Pone a cero la memoria.
- `void bcopy(void*, void*, int)`: Copia memoria.
- `void* memccpy(void*, void*, int, int)`: Copia memoria hasta que encuentra un *byte*.
- ...



# Entrada Salida

---

- **int**     **fprintf(FILE\*, ...)**: Salida sobre fichero.
- **int**     **fscanf(FILE\*, ...)**: Entrada desde fichero.
- **int**     **sprintf(char\*, ...)**: Salida sobre un buffer.
- **int**     **sscanf(char\*, ...)**: Entrada desde un buffer.
- **int**     **fgetc(FILE\*)**: Lee un carácter desde fichero.
- **char\*** **fgets(char\*, int, FILE\*)**: Lee una línea.
- **FILE\*** **fopen(char\*, char\*)**: Abre un fichero.
- **int**     **fclose(FILE\*)**: Cierra un fichero.
- **int**     **fflush(FILE\*)**: Descarga un buffer de fichero.
- **int**     **feof(FILE\*)**: Indica si ha finalizado un fichero.

# Ficheros Especiales

---

Existen tres variables de tipo FILE\* asociados a tres ficheros estándar:

- **stdin**: Entrada estándar.
- **stdout**: Salida estándar.
- **stderr**: Salida de error estándar.

```
fprintf(stdout, "Usuario: ");
```

```
fscanf(stdin, "%s", usr);
```

```
fprintf(stderr, "Error: No válido");
```

# Ordenación y Búsqueda

---

- `void* bsearch(void*, void*, int, int, int (*)(void*, void*)):`

Búsqueda binaria sobre lista ordenada.

- `void qsort(void*, int, int, int (*)(void*, void*)):`

Ordenación mediante el algoritmo *quicksort*.

```
char *vec[10]={ "casa", ..... };
```

```
qsort((void*)vec, 10, sizeof(char*), strcmp);
```

# Conversión de Tipo

---

- **int**     **atoi(char\*)**: Traduce de string a entero.
- **long**   **atol(char\*)**: Traduce de string a un entero largo.
- **double** **atof(char\*)**: Traduce de string a real.
- **long**   **strtol(char\*,char\*\*,int)**: Traduce de array a entero (en diferentes bases).
- **double** **strtod(char\*,char\*\*)**: Traduce de array a real.
- **char\***   **itoa(char\*,int)**: Traduce un entero a array.

Como alternativa se pueden usar las funciones:

**sscanf** y **sprintf**.

# Funciones Matemáticas

---

- `double exp(double)`: Calcula  $e^x$ .
- `double log(double)`: Calcula el logaritmo natural.
- `double log10(double)`: Calcula el logaritmo en base 10.
- `double pow(double, double)`: Calcula  $x^y$ .
- `double sqrt(double)`: Calcula la raíz cuadrada.
- `double sin(double)`: Calcula el seno (en radianes).
- `double cos(double)`: Calcula el coseno (en radianes).
- `double sinh(double)`: Calcula el seno hiperbólico.
- `double atan(double)`: Calcula el arco tangente.
- ...

# Uso de Funciones de Librería

---

El uso de las funciones de librería estándar esta sujeto a las siguientes recomendaciones:

- Estudiar la página del manual (**man 3 sprintf**).
- Incluir en el fichero fuente el fichero de cabecera adecuado (**#include <stdio.h>**).
- Enlazar la librería si es necesario (**gcc .... -lm**).

---

# Programación en C

## Argumentos del Programa

# Argumentos de `main`

---

La función principal `main` puede recibir argumentos que permiten acceder a los parámetros con los que es llamado el ejecutable.

```
int main(int argc, char* argv[ ])
```

- `int argc` : Número de parámetros.
- `char* argv[ ]` : Parámetros del ejecutable.



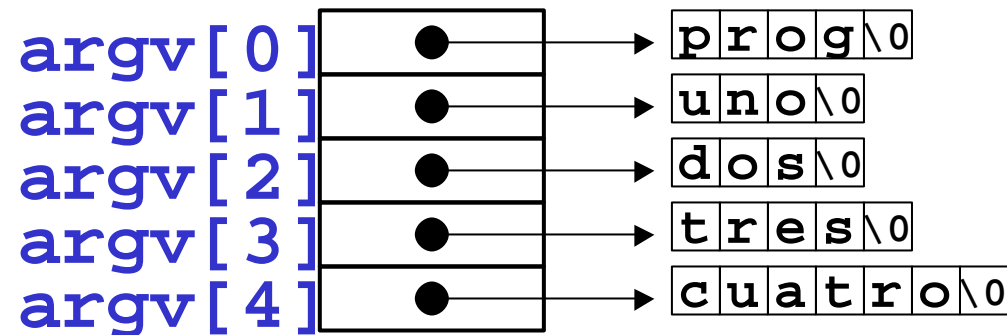
# Argumentos de main

---

```
$ gcc prog.c -o prog
```

```
$ prog uno dos tres cuatro
```

```
int main(int argc, char* argv[])  
    argc=5
```



# Argumentos de `main`

---

```
int main(int argc, char* argv[])
{
    int i=0;
    printf("Ejecutable: %s\n", argv[0]);

    for(i=0; i<argc; i++)
        printf("Argumento[%d]: %s\n", i, argv[i]);

    return 0;
}
```